NAME ASSIGNMENT IN COMPUTER NETWORKS

Technical Report 1080-310-1

October 1982

By: Raphael Rom, Research Engineer

Telecommunications Sciences Center
Computer Science and Technology Division

This research was sponsored in part by the Minna Jame Heinemann Stiftung, Germany and by the Defense Advanced Research Projects Agency under ARPA Order No. 2303, Contract No. MDA-903-80-C-0222, monitored by Dr. B. Leiner

SRI Project 1080

SRI International
333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 859-6200 • TWX: 910-373-2046 • Telex: 334 486

022

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A123 854	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Name Assignment in Computer Networks		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER 1080
7. AUTHOR(s) Raphael Rom, Research Engineer		8. CONTRACT OR GRANT NUMBER(s) MDA903-80-C-0222
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, California 94025		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Code No. P62708E
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE October 1982
		13. NUMBER OF PAGES 34
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Name assignment, network merger, network startup, duplicate name resolution

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Name assignment, the procedure by which an identifier is assigned to the nodes of a network to be used subsequently by other networking mechanisms, is investigated in detail in this report for such situations as node restart and relocation, network merger and startup. Algorithms for a four-phase approach to network realocation are proposed. In the first phase, pairs of neighboring nodes get acquainted; in the second phase the pair must decide

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

whether their networks are indeed distinct and should merge; in the third phase each of the two nodes alerts th other nodes in its subnetwork of the prospective merger and proceeds to phase four--the merger itself.

CONTENTS

I. INTRODUCTION

Name assignment is the procedure by which an identifier is assigned to the nodes of a network to be used subsequently by other networking mechanisms. The problem of name assignment has not been properly dealt with in spite of the great advances in computer networking techniques, and in spite of the wide attention that naming conventions have received[1,2,3]. To date, most algorithms requiring node identification assume that the names they need have been preassigned. Moreover, many algorithms[4,5] assume that a unique name assignment is guaranteed.

In most networks names are assigned administratively--that is, by a network operating authority rather then by the network itself; this is usually done on a long term basis. Where name assignment is done dynamically, such as in the packet radio network[6], a centralized approach is being used.

In military networks where survivability is a major issue, the name assignment problem being the heart of the reconfiguration problem[7], becomes more acute. In mobile packet radio networks, where the network can be partitioned and reconstituted as a result of topological changes or the deployment of airborne packet radios, or when the size of the network becomes very large, name assignment becomes a matter of being able to operate at all[8,9].

In the following sections we investigate the name assignment problem in full detail. The body of the paper contains only a descriptive presentation of the algorithms, while details are reserved for the appendices.

II. BASIC APPROACH

In this paper we use the term network to denote a collection of communicating entities interconnected by links in such a way that there exists a path between each two entities and within which data destination can be uniquely specified. Network topology is not necessarily fixed in time--neither are the number of entities and links nor their relative interconnection. (In this paper we shall use the terms "node" and "entity" interchangeably; the term "subnetwork" will denote a network that is about to merge with another.)

This definition is intentionally broad. It allows us to look at the naming problem from a more abstract standpoint. For example, by considering packet switches of a communication network as the entities, our notion of a network matches the common definition. In another example, if the entities are cluster controllers[9], naming deals with the management of a higher level configuration. Thus "Internet" in the ARPA taxonomy[10] and "Group" in MINA taxonomy[7] are both networks according to this definition.

Name assignment is the process of ascribing an identifier to each member of a set of entities in a way that guarantees uniqueness within that set. The names we ascribe are not globally unique, but have a limited scope both in time and space. However, they are guaranteed to be unique in the domain in which they are generated and assigned.

The name assignment problem manifests itself in several situations:

1. Node restart. A new node needs to join an already operating network after being inoperative for some period of time. This node must be assigned a unique name to be used subsequently for referencing that node, and its existence must be made known to (all) other network nodes.
2. Node relocation. A node heretofore belonging to a network moves to another operational network. This is a typical case in packet radio networks.
3. Network merger. Two networks, heretofore operating separately, wish to merge and become a single network operationally.
4. Network startup. A network that has been completely inoperative is being started. All nodes of the network must be assigned names.

We propose a simple approach for name assignment procedure which covers all the above cases. A node that does not have a name, such as one that just became operational, chooses a name for itself and starts looking for neighboring nodes to join them into a single network. When

the new node becomes acquainted with its neighbor, it is assigned a new name that is guaranteed to be unique within the newly formed network. Node relocation is similar except that the affected node can use its old name.

The same approach is used for the case of network merger. The process starts when two neighboring nodes become acquainted and realize they belong to two different networks. These two nodes then coordinate the merger of the networks, i.e., reassigning names such that within the merged network names will be unique.

Network startup is a combination of the above two cases. Nodes start by assuming a name for themselves and then join to form clusters that gradually merge to form the final network. It should be noted that network startup can (and probably does) start concurrently at several nodes.

We propose a four-phased approach to name assignment. In the first phase pairs of neighboring nodes get acquainted. In the second phase the pair must decide whether their networks (or clusters) are indeed distinct and should merge. In the third phase each of the two nodes alerts the other nodes in its subnetwork of the prospective merger and proceeds to phase four--the merger itself. An additional cleanup phase might be added to perform network-specific operations before the name assignment is over.

The algorithms used to achieve our goals are based on solutions to two classical problems: mutual exclusion and leader selection. The mutual exclusion algorithm proposed by Dijkstra[11], and for which several solutions exist[12,13], is generalized to perform a mutual exclusion for groups. Our election algorithm is based on those described in the literature[14,15,16], but is generalized to cover several concurrent election algorithms that must be coupled.

The described algorithms are all distributed. Survivable networks should not centralize any of their activities to avoid a single point of failure. But that does not mean that the entire set of algorithms must be distributed. One can, for example, adopt the approach that within each network a leader is chosen distributively (by means such as described in [15]) to coordinate name assignment thereafter. The deficiency of this approach is the need for acquisition of status information by a new leader when it is first elected. Also, as is pointed out later in this paper, some of the problems faced by a distributed algorithm are not eliminated in a centralized environment.

We describe our procedure for radio networks in which each node is assumed to have a unique ID and a name. Unique IDs are universally unique and therefore positively distinguish all nodes from one another. Unique IDs are used for authentication purposes only. Names, on the other hand, are completely unrelated to unique IDs and are used for data

destination specification, i.e., they are used in packet headers. Unique IDs are not used as names because the latter are likely to be much shorter and usually have internal structure that is instance-dependent.

The entire process starts when a node detects a new neighbor. The detection mechanism itself is not specified. For example, a node can be actively looking for a neighbor when it first comes up, or it can examine the "I am alive" messages that are exchanged in radio networks to establish network connectivity, to identify new neighbors. Once a new neighbor is detected, the four-phased approach commences.

III. THE FOUR PHASES OF NAME ASSIGNMENT

A. Phase 1: Getting Acquainted

This phase starts when a node detects a neighbor suspected of belonging to a different network. In this phase we assume that all the suspected new neighbors are indeed from different networks, leaving the final resolution of that fact to the next phase. Since many nodes may concurrently and independently detect neighbors, we consider here all those nodes that have detected a potential new neighbor, as well as all the potential neighbors themselves.

The purpose of this phase is to order these nodes in pairs so that at least one pair is formed, the two nodes constructing a pair know each other, and every node that does not belong to any pair is positively notified so that deadlock will be avoided.

The get-acquainted (GA) algorithm described in Appendix A is appropriate for that purpose. It is a special type of matching algorithm. While most matching algorithms attempt to arrange a perfect matching, the GA algorithm attempts to arrange only as many node pairs as possible--with just one pair guaranteed (but many likely). Furthermore, most matching algorithms require that all participating nodes know one another, whereas the GA algorithm operates in an environment in which each node knows only its neighbors.

The full detail of the algorithm is given in Appendix A; stated briefly it proceeds as follows. Participating nodes send a message, each to its chosen neighbor, indicating that node's wish to get acquainted. The message contains the node's unique ID, which is used to reconcile conflicting requests and to form a partial ordering among the nodes. Two types of messages are used: **REQUEST** and **REJECT**, with the **REQUEST** message serving also as acknowledgment. Two nodes that have successfully exchanged **REQUEST**s are considered to have formed a pair. A **REJECT** message is sent by a node in response to a **REQUEST** message when it is clear that the two will not constitute a pair, e.g. when the node has already chosen a potential partner, but has not received final confirmation.

Obviously, if two nodes send a **REQUEST** to one another concurrently, each interprets the other's message as an acknowledgment and they will thus have formed a pair. More complex situations are also handled such as when the chosen neighbor has itself initiated a request to another one, or when a loop of **REQUEST** messages is created that may lead to a deadlock.

B. Phase 2: Determining Network Affiliation

Having made each other's acquaintance, the pair of nodes must now decide whether or not they belong to the same network. Obviously, if they have differenct network names they belong to different networks. The difficulty arises when the nodes have the same network name--in which case nothing can be said regarding their affiliation on the basis of network names alone. Figure 1 illustrates this problem by depicting two nodes, 1 and 2, both belonging to a network whose name is A. In one case they belong to the same network, in the other they do not.

To resolve this complication, characteristics other than network name must be considered. Let us assume that nodes A1 and A2 have become acquainted and that the corresponding network names are the same. A1 then checks whether a node whose name is A2 exists in its network (this is an internal check that does not necessitate any message exchange); if such a node does not exist, A1 concludes that its partner belongs to a different network. A2 performs an identical check. This check is not decisive, however, since it is possible that the two belong to two different networks, each with both an A1 and A2 node.

The 'local interrogation scheme' resolves such conflicts. A1 sends an interrogation message to the node named A2 in its own network asking whether it has recently become acquainted with another node whose name is A1 and whose unique ID is that of A1. A positive response means that the two belong to the same network and a negative response means they do not.

The local interrogation scheme provides a decisive answer to the affiliation question at the cost of two additional messages. It is possible to get a partial answer without the message exchange. For example, the acquainted nodes can exchange (in the getting-acquainted phase)

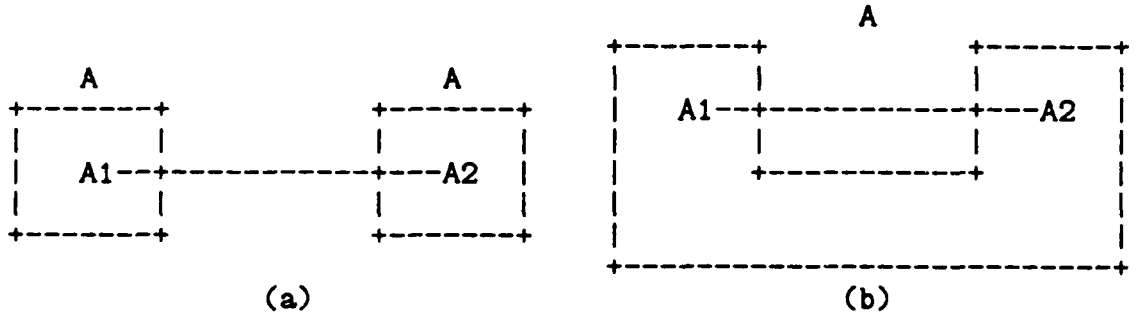

Figure 1: Determining Node Affiliation
(a) Nodes belonging to different networks
(b) Nodes belonging to thesame network

such characteristics as network size, intranetwork name, and a name bit pattern. A name bit pattern (NBP) is a bit pattern having the ith bit 1 if name i is assigned to a node in the network and 0 otherwise. If the sizes or the NBPs are different, the nodes belong to different subnetworks, as is the case when the two nodes happen to have the same intranetwork name. (It should be pointed out that the NBPs are further used in Phase 4.) Since these characteristics may not yield a positive resolution, nodes may still use the local interrogation scheme when all else fails.

If either of the nodes determines that a merger should take place, it sends a LETS-MERGE message to its partner. Once each partner has sent and received such a message, both proceed to phase 3.

C. Phase 3: Electing Representatives

Having gotten acquainted with one another and deciding in favor of a merger, the two nodes coordinate in phase 3 all necessary activities before the actual merger takes place. Since many node-pairs may independently and concurrently do likewise, they must all become aware of one another. Figure 2 illustrates the problem.

In Figure 2a, nodes A1 and A2 in network A have just completed Phase 1 with nodes B1 and B2 of network B. Thus two pairs exist: A1-B1 and A2-B2, both attempting to merge the networks A and B. Since both activities need not proceed simultaneously, one representative of each network must be chosen.

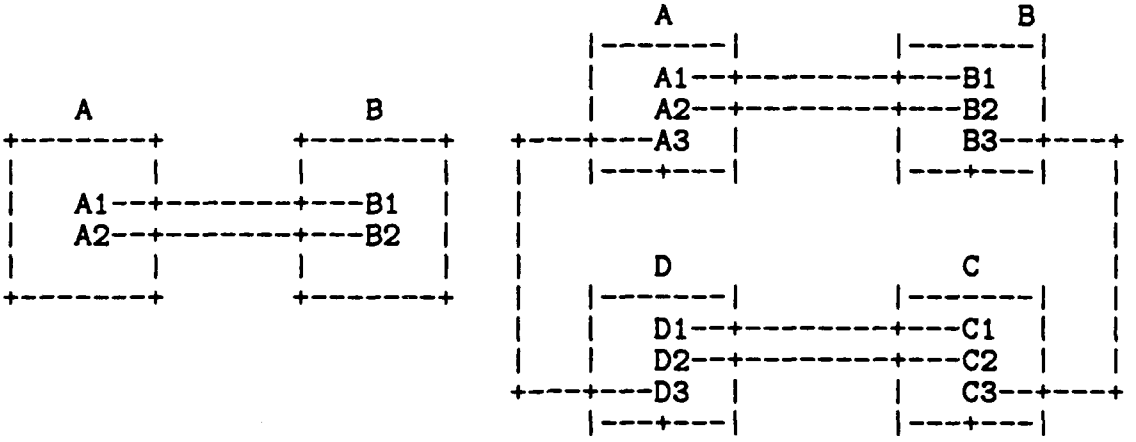

Figure 2: The Need for Node Coordination
(a) Two networks
(b) Several networks

A simple election process in each network of the type described in [15] is inadequate for two reasons. First, a deadlock will occur if A1 is elected in network A while B2 is elected in network B. Second, the members of the group in question (A1, A2, B1, B2) do not all know one other as is typically required. This is the case in which two independent yet tightly coupled elections must take place.

Yet another situation is depicted in Figure 2b. Here a more complex deadlock may occur, involving more than two subnetworks; this means that a pair of networks must be decided upon before representative nodes are chosen.

As the first step toward overcoming these deadlock possibilities, a group mutual-exclusion (GME) algorithm is executed in each network separately. At the end of execution each node knows whether it should proceed, and if so, who else does.

The GME algorithm which resembles the mutual exclusion algorithm of[13] proceeds as follows. Two messages are used: **REQUEST** and **ACK**. A node wishing to be selected (i.e., to enter the **REQUEST** message containing a token. A node receiving a **REQUEST** replies with an **ACK** unless it has previously sent a **REQUEST** or is in its critical section. The elected group will consist of all those nodes that have received responses from all others. The token is used to further subdivide the group into subgroups. The **REQUEST** message may also be used to exchange information among group members, to be used in subsequent steps. Full details of the algorithm is given in Appendix B.

For our purposes the token used by each node is the network name of its foreign neighbor. In each subnetwork the elected group therefore consists of all nodes that are neighbors to the highest numbered subnetwork. All participating nodes <u>not</u> within the group notify their foreign neighbors and postpone all further merger activity until some later time.

Alternatively, one may use the pair (network-size, network-name) as a token to ensure that merger between the two largest networks will take place; network name is used to discriminate among equal-sized networks. When more than one network name is involved, a token can be chosen that, in coordination with the neighboring subnetworks, guarantees exactly one leader per subnetwork.

Consider now the group of all nodes in the various networks that have been elected in the GME algorithm. It consists of an unknown number of nodes belonging to an unknown number of networks with at most two network names involved. Each node has a link to a node of another network (its original neighbor). Figure 2b depicts such a situation with 12 nodes in 4 networks in which A=C and B=D; another possible situation would involve only one network name (i.e., A=B=C=D).

Next, one of the existing pairs must be chosen. Our approach is to choose a single leader from the group that, with the aid of its partner, will become the merging coordinators in the next phase. This is done via the loop-based-tree mutual exclusion (LBT-ME) algorithm described next.

Consider a graph created by the nodes in question with directed arcs constructed as follows. Within each network every participating node creates an arc to the node with the largest intranetwork name (which is unique). The node with the largest intranetwork name creates an arc to its partner in the foreign network. The resulting graph consists of several subgraphs, each of which is a loop-based tree--i.e., a loop to which all the rest of the nodes are connected in a treealike manner; if any of the loop arcs is removed, a tree with directed arcs remains. Figure 3 shows a loop-based tree.

Messages are sent along the arc's direction and include the originator's unique ID. Each node keeps track of the highest ID observed by it, and only messages with higher IDs are passed along the graph; the rest are dropped. A node receiving its message back is the leader. The algorithm is described in detail in Appendix C. The chosen leader than notifies its foreign neighbor and both become the coordinating nodes for the merger that takes place in the next phase.

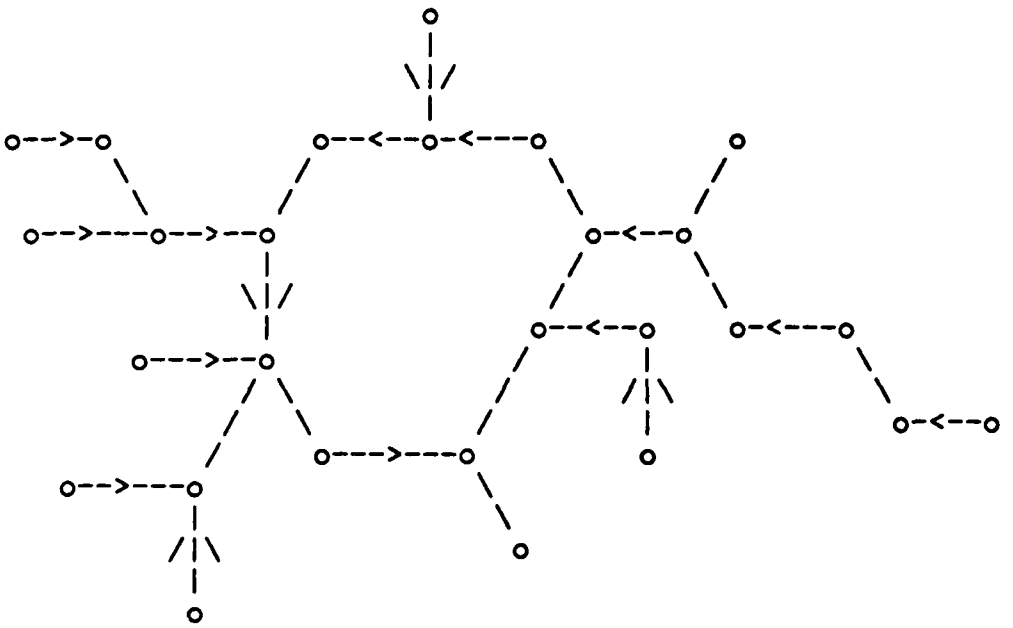

Figure 3: A Loop-Based Tree Topology

Note that the GME executed previously is necessary to assure that within each network all nodes that completed phases 1 and 2 are aware of one another, as required by the LBT-ME algorithm.

D. Phase 4: Network Merger

At the beginning of phase 4 there exist two neighboring nodes, one from each subnetwork, that have been chosen to monitor the merger that is performed in this phase. These nodes, which we call merger-coordinating nodes, send to one another a MERGE-REQ message to indicate their readiness. When each node has both sent and received such a message, the two subnetworks merge, i.e., names may be reassigned so that all names within the combined network will be unique.

The actual assignment of names to the nodes of the combined network cannot be discussed in general since names often have environment-specific structure. In the following discussion we shall examine a method for node renaming in a flat name space.

The simplest statement of the problem is this: given two sets of nodes A and B each with unique node names Na_i and Nb_i, respectively, one needs to construct a new set of nodes C=A U B with names Nc_i such that these names are unique. A practical solution must be an algorithm that terminates very fast, for all other data transmission services must be suspended when a network is being renamed. To expedite the termination of renaming it is desirable to

- o Provide a renaming scheme that is computationally simple and requires very little message exchange.
- o Minimize the number of nodes that are actually renamed.
- o Be able to perform the renaming in parallel with several starting points.

A simple way to achieve the above goals is the following. The larger of the two networks, say B, is left untouched--that is, all its nodes retain their intranetwork name as well as their network name. Nodes in network A whose intranetwork name is not in use in network B retain their intranetwork name (the network name will be that of B). intranetwork name. These are deduced from a name bit pattern that is broadcast by merging coordinator.

Formally, let NBPA and NBPB be A's and B's bit pattern (assumed known by all nodes in A). Node i performs the following:

```
if NOT NBPB[i] then
        count:=0;
        for k:=1 step 1 until i do
                if (NBPA·NBPB)[i] then count := count+1;
        newname:=1;
        for k:=1 step 1 until count+1 do
                while (NBPA+NBPB)[newname] do newname := newname+1
else
        newname:=i;
```

Upon completion 'newname' is the new intranetwork name for node i.

This algorithm is simple, can be performed in parallel by all nodes and requires the transmission of only one message per node of network A. Moreover, every node can compute every other node's new name, so that packets in transit need not be retransmitted; their destination specification must be changed instead.

E. Notes

The algorithms described in the previous subsections have a particular feature in common: they are time-independent, that is, time plays no role in any of the steps. This does not mean that time should not be used. For example, it may be advisable to use time-outs to verify that nodes have not crashed in the midst of a critical operation, such as in the middle of phase 3, leaving the network in an undesired state. The time independence of the algorithms themselves makes the use of timeouts for the purpose of improving robustness more powerful.

These algorithms work remarkably fast for simple configurations. For example, when only two nodes are involved (a frequent situation) the GA and LBT-ME algorithm each require one message from each node. The renaming and GME algorithms require one and two messages respectively, per subnetwork node.

Regular data delivery can proceed normally during phases 1, 2, and 3, as well as during part of Phase 4 provided nodes are properly coordinated. The only 'dead period' is between the time the node gets the renaming message until it completes the computation of the new names. To remain synchronized with respect to names, nodes must be able to interpret the names in all incoming messages properly. This can be done by inspecting the network name or by using phase numbers for each renaming cycle.

All these algorithms work similarly in wired point-to-point networks, since no particular use of the medium is made. It should be pointed out, though, that in broadcast networks the GME algorithm can be implemented more efficiently.

Finally, a word about centralized systems. Here we assume the existence of a 'name assignment center,' responsible for assigning names to nodes. Such a configuration does not change the nature of phase 1--the GA algorithm must be used to elect a pair of assignment centers from two neighboring networks. A modified phase 2 is necessary if one assumes that the center does not (or cannot) store all the unique IDs of the nodes under its jurisdiction. Phase 3 can be completely eliminated as all decisions are made within the center. Phase 4 remains unchanged with the name assignment centers playing the role of the merger coordinating nodes.

F. Topologies Without Unique IDs

The lack of unique IDs manifests itself in all phases of the name assignment process. In fact, in an environment where nodes cannot be positively distinguished from one another, name assignment cannot be accomplished. That is, the uniqueness of names cannot be guaranteed within any given set of (more than two) nodes.

Consider, for example, the topology depicted in Figure 4, where two nodes named A and two named B are positioned such that each A hears both B's but not the other A, and each B hears both A's but not the other B. As a result A will not be able to distinguish between the two B's and, since there is a nonzero probability that both B's will generate the same sequence of random numbers, this situation will persist.

Unique IDs are used for purposes of authentication and ordering, neither of which can be provided absolutely in this environment. The crux of the problem is the fact that the nodes need to conduct dialogues with single destinations. It is insufficient for the nodes to verify at the beginning of the dialog that they are the only two partners of a

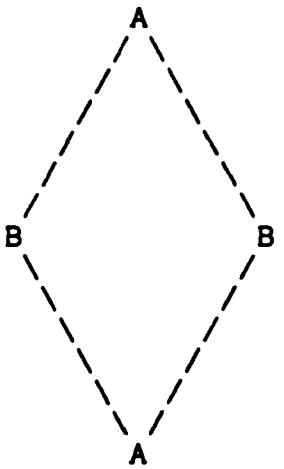

Figure 4: Deadlock situation in radio networks

conversation. This activity must go on continuously for the duration of the conversation as another node my, at random, assume the same ID as one of the original participants.

Name assignment can, however, be accomplished with an arbitrarily high probability of success by choosing IDs from a large domain or by repeating crucial steps of the algorithm several times each with a newly selected ID. The appendices elaborate on the difficulties caused by the lack of unique IDs in any one of the phases.

Practically, name assignment in an environment without unique IDs seems too complex to be worthwhile implementing. If reliance on a built-in unique ID in every node is to be avoided, mixed mode could be considered. There are two mixed-mode possibilities:

1. Using semiunique IDs--that is, IDs composed of two fields, one of which is constant and highly likely (but not guaranteed) to be unique, while the other is a random number chosen dynamically by the node. Thus, the algorithms will have to be repeated only a small number of times (if at all) to guarantee success with a high probability.
2. Using two kinds of nodes, some with an absolutely guaranteed unique ID and some without. Those with unique IDs will be able to participate independently in all phases and provide a reference for those without unique IDs. The latter will have to consult the former before critical steps are taken.

APPENDICES

A. THE GET-ACQUAINTED (GA) ALGORITHM

1. Definition

A get-acquainted algorithm is one that arranges the participating nodes in pairs while ensuring that:

- o At least one pair is formed
- o Each participating node knows whether or not it is one of a pair
- o Two nodes comprising a pair know each other.

2. Description

The GA algorithm we propose starts by having some of the participating nodes send a request to a node of their choice. A directed graph is thus constructed, with the arcs directed from the originator of the request to its destination. The graph thus formed is not necessarily connected. The topology of each of the connected subgraphs is either linear or a loop-based tree. (A loop-based tree is a topology in which at most one directed loop may exist; if one arc of the loop is removed, a tree is left.)

The arcs represents bidirectional communication links. Because there is no communication among the subgraphs the algorithm operates on each of the subgraphs independently. The description that follows assumes, without loss of generality, that the entire graph has the loop based tree topology.

Loop-based-tree topologies are reduced even further by having each node select one of the arcs pointing toward it, and deleting all the rest. The loop based tree is then partitioned into a set of graphs containing at most one loop and perhaps several linear configurations. Within each of these graphs we form pairs from adjacent nodes. IDs are used to break a deadlock if a loop is constructed.

3. Specification

Each participating node must obey the following conditions:

C1. Each node chooses at most one successor and one predecessor.

C2. A pair is formed when two nodes receive nonforwarded requests from each other.

In addition, each node operates according to the following rules:

R1. Messages received from neither the predecessor nor the successor are rejected.

R2. A node with a predecessor but no successor forms a pair with its predecessor (by sending it a request).

R3. For as long as a node is not included in any pair, it forwards to its successor all requests it receives from its predecessor that originated at a node with a higher ID.

R4. A node rejects its predecessor when

(a) It has formed a pair with its successor.

(b) It has received a message originated by itself.

4. Proof of Features

Feature 1: At least one pair is formed.

As a result of Condition C1, each node deals with only two nodes--its predecessor and its successor. Each node can therefore be part of either a linear or circular configuration.

In a linear configuration, there is one node with no successor. According to rule R2, this node will form a pair with its predecessor.

In a circular configuration, the request originating at the node with the highest ID is forwarded by all nodes (according to rule R3) until it arrives at its originator. According to rule R4b, the originator rejects its predecessor--which then becomes a node without a successor and subsequently forms a pair in accordance with rule R2.

In the degenerate case in which only two nodes are involved, the original exchange of requests immediately forms a pair (Condition C2).

[]

Feature 2: Each member of a pair knows the other.

This is trivially true by virtue of Condition C2. []

Feature 3: Each request is answered.

As a result of Feature 1, at least one pair is formed. We now consider the successor and predecessor nodes of this pair.

According to rule R4a the predecessor node is obviously notified. This node then becomes one with no successor, in which case rule R2 applies (as long as this node itself has a predecessor). The formation of pairs thus propagates in the predecessors' direction.

Now observe the successor of the pair. The pair could have been formed only if,at that time, it had no successor (rule R2). This could happen if the pair never had a successor or if the successor sent a rejection message. Rejection, in turn, is sent only by a node that has formed a pair (rule R4a) or by a node has detected a circular configuration (rule R4b) that still has a successor from which a message will eventually arrive. []

Feature 4: The GA algorithm terminates.

The algorithm terminates when no more messages are in transit. Feature 3 assures that each node knows whether it is single or part of a pair, and will therefore cease generating messages. The only messages still possibly in transit are those being forwarded; according to rule R3, however, these will be ignored and stopped upon arrival at a node that belongs to a pair (of which, according to Feature 1, there is at least one). []

It can easily be shown that the above rules and conditions address all the messages that might be received:

- o Messages received from neither the predecessor nor the successor are covered by rule R1
- o Requests from the predecessor are covered by rule R2 or R3 depending on whether the receiving node has a successor.
- o Condition C2 covers requests received from the successor.
- o Rejections received from the successor are covered by rule R2
- o A rejection from the predecessor cannot arrive since rejections are sent only to predecessors (rule R4).

Since the treatment of all possible messages is specified, and according to Features 1 through 4 our goal is achieved, the GA algorithm is correct as long as there exists a node to issue the first request.

5. Specification--Take 2

The GA algorithm can be equivalently specified by the following Algol-like code:

```
INITIALIZATION
    successor = 0
    predecessor = 0
    tail = FALSE

Local request(destination)

 1: if successor != 0  then
 2:     send(destination, REQUEST)
```

```
 3:      successor = destination

Receive REQUEST(from, orig)

 4: if successor = 0 then %Original request %
 5:      send(from, REQUEST(myid, myid) ) %send ACK%
 6:      exit(from)
 7: else if successor = from then %this is an ACK%
 8:      if predecessor != 0 then send(predecessor, REJECT)
 9:      exit(successor)
10: else % successor NEQ 0 %
11:      if predecessor = 0 then % first request %
12:          predecessor := from
13:          if orig > myid then send(successor, REQUEST(myid, orig))
14:      else  % predecessor NEQ 0 %
15:          if from != predecessor then
16:              send(from, REJECT)
17:          else % from = predecessor %
18:              if orig > myid then
19:                  send(successor, REQUEST(myid, orig))
20:              else if orig = myid then %got my msg back %
21:                  send(predecessor, REJECT)
22:                  tail := TRUE
23:              else % orig < myid %
24:                  NIL


Receive REJECT

25: if tail then
26:      exit(0)
27: else
28:      send(predecessor, REQUEST(myid, myid) ) %this is an ACK%
29:      exit(predecessor)
```

In the foregoing, predecessor and successor are the corresponding IDs (0 is assumed to be an illegal one), and tail is a boolean variable indicating the detection of a loop. In addition, the above specification meets all conditions and rules:

- o condition C1 is met in lines 1-3 for the successor and 11-12 for the predecessor.
- o Rule C2 is met in lines 5, 9, and 28. (In these lines it is shown that REQUEST messages have been directly exchanged between two adjacent nodes).
- o Rule R1 is obeyed in lines 15-16
- o Rule R2 is obeyed in lines 5 and 28
- o Rule R3 is obeyed in lines 13, and 18-19
- o Rule R4a is obeyed in line 8 and rule R4b in line 21.

The if-else nesting demonstrates that each message is treated exactly once and that all cases are covered.

6. Configurations Without Unique IDs

The algorithm must be adapted slightly to operate in an environment where unique IDs are not available. The main difficulty is that of abiding by condition C1, since a node may not be able to determine if it has more then one successor or more then one predecessor.

With regard to the rules there is clearly no problem in obeying rule R1. Obeying rule R4 is proper, as it is intended to reject all predecessors (in the original algorithm only one existed). Conformance to Rule R3 may cause inefficiency if a node has more than one successor; in this case, forwarded messages propagate to branches they do not have to, but do not cause any harm thereby.

Rule R2 is the only one needs to be adapted because one must ensure that a given node is not a partner in more than one pair. If a node has two (or more) successors and if either of them attempts to form a pair with their common predecessor, ambiguity results (obviously there is no problem if both reject). A similar problem occurs when a node determines it has to form a pair with its predecessor, but cannot direct its response to only one (and may not know it has many).

A possible solution is an authentication step once pairs have been formed. In this step a sequence of random numbers is exchanged between the parties (a node with its successors or a node with its predecessors) with the hope that they will eventually generate different numbers and become distinguishable. If random numbers are equally distributed with p being the probability of selecting any given number, then the probability of k nodes generating the same random numbers in r attempts is $p^{r(k-1)}$ and the total probability of failure is bounded by $\frac{p^r}{1-p^r}$. This probability obviously approaches 0, and faster the larger the domain of random numbers.

B. THE GROUP MUTUAL EXCLUSION ALGORITHM

1. Description

Let there be a network with N processes (or nodes), all known and reliably accessible to one another (either directly or indirectly). The GME algorithm distributively selects a group of processes that concurrently try to achieve the same goal. The problem is identical to that of mutual exclusion but does not impose the constraint that exactly one process must be in the 'critical section'. Instead it requires that

- o A group of processes wishing to enter the critical section can do so only if all processes that have previously been in the critical section have already left it.
- o All processes that form a group (i.e., are entering the critical section together) must know one another.

The approach to a solution (and therefore to the proof) is similar to that of[13]. Time is divided into sequentially numbered cycles. A process wishing to join the group sends a request message to all members of the group (possibly utilizing a broadcast facility). The request message contains the cycle number and a token, that is used subsequently to determine which processes actually form the group. If all processes use the same token, all requestors join the group.

A process receiving a request acknowledges it immediately unless it is attempting to join the group itself, i.e., has transmitted a request in this cycle, in which case it defers any reply.

A process that has issued a request will eventually receive a message (in the current cycle) from each of the other N-1 participants--a request from each process attempting to enter the group and acknowledgments from those that are not. This part of the algorithm terminates when all members of the group have received a response.

The next step is a distributed decision as to who should remain in the group. This can be done by using information already transmitted, such as the token, or by an explicit exchange of messages among the group members. Finally, to proceed to the next cycle, a COMPLETE message must be sent, signifying that all those in the group have left their critical section.

2. Specification

The algorithm is presented now in an Algol-like language, using send and receive processes that run asynchronously and share data.

```
SHARED DATA

        integer cycle_num INIT 1
        boolean req_lock INIT FALSE
        boolean array current, next
        constant integer myid, N

SEND PROCESS
    % invoked when this node wishes to enter critical section %

S1:     if req_lock then next[myid] := TRUE
S2:     waitfor( req_lock = FALSE)
S3:     transmit_REQUEST(myid, cycle_num)
S4:     req_lock := TRUE
S5:     current[myid] := TRUE
S6:     reply_count := N-1
S7:     waitfor(reply_count = 0)
        % critical section here possibly including
S8:         transmit_COMPLETE(k) %
S9:     RESET

RECEIVE PROCESS
    % Invoked when a message arrives %

        REQUEST(id, num)

R1:             if num > cycle_num then next[id] := TRUE
                else
R2:                 current[id] := TRUE
R3:                 if NOT current[me] then send_ACK(myid, id)
R4:                 req_lock := TRUE
R5:                 reply_count = reply_count-1

            ACK(id)

R6:             reply_count = reply_count-1

            COMPLETE(k)

             if NOT current[myid] then
R7:                 reply_count := reply_count+k
R8:                 waitfor(reply_count=0)
R9:                 RESET
```

```
RESET
    % Invoked explicitly %

        increment cycle_num
        current := next
        clear next
        if current[myid] then
            req_lock := FALSE
        else
            for j:=1 step 1 until N do
                if current[j] then
                    send(ACK, j)
                    reply_count := reply_count-1
            req_lock := (reply_count<0)
```

3. Discussion

In the above specification

- o The 'transmit' and 'send' functions cause messages to be sent; the former to all other processes and the latter to the specified destination.
- o The number k contained in the COMPLETE message indicates the number of group members in the specified cycle.
- o The 'increment' construct performs a modulo C increment. It is shown in the following that modulo 2 counting suffices.
- o The req_lock flag serves to ensure that at most one **REQUEST** is sent per cycle from any given source.
- o The ith entry of the current and next arrays records whether or not process P_i belongs to the group of the current and next cycles, respectively.

The COMPLETE message causes the process to wait until all k requests of that cycle have arrived. The algorithm does not specify which process sends the COMPLETE message, but requires that only one message per group be sent. Note that if this process does not participate in this cycle, its reply_count is negative, since it started with 0 and was decremented during the process. In this state the reply count will therefore reach 0 from below.

The cycle ends when all expected messages have arrived. At this time the ith entry of the current array is TRUE if P_i is a member of the group and FALSE otherwise. Before proceeding, the state is reset by making the next array the current one, transmitting a **REQUEST** or sending **ACK**s to all those to whom it was deferred, and possibly releasing the lock.

Message traffic is as follows. Assume k processes out of all N wish to enter the critical section. Each of the k processes broadcasts

a request to all others, resulting in k(N-1) messages; each of the other N-k processes sends an **ACK** to each of the k group members resulting in k(N-k) additional messages. Finally one of the group members sends a COMPLETE message to the N-k nonmembers. Thus a total of $2k(N-1)+N-k^2$ messages is sent.

The algorithm does not enforce fairness. One could, as part of RESETting, ensure that a process that participated in the current group will not participate in the next one. Also, if the number of group members is limited, those elected need to decide--in some fair way--who stays and who does not, so that no process will be permanently excluded.

4. Proof of Features

The following observations can be made:

- o RESET is called only once in every cycle, at the point where the process perceives the end of cycle, i.e., when all expected messages have arrived (lines S8, R9).
- o RESET is the only place where cycle_number is incremented and where the req_lock may be set to FALSE.
- o The entry 'current[myid]' is set only after a **REQUEST** has been transmitted.

Feature 1: Messages are exchanged between two processes only if at least one of them attempts to enter the critical section.

REQUESTs are sent only if a process attempts to enter the critical section; **ACK**s are sent only in response to a **REQUEST** (line R3). []

Feature 2: Within each cycle at most one message is sent from any process to any other process.

We observe the generation of both **REQUEST** and **ACK** messages. These are governed by the two booleans req_lock and current[myid] correspondingly. Both booleans are being reset only once per cycle--during RESET.

The transmission of a **REQUEST** causes the setting of both req_lock and current[myid] (lines S4, S5) to preclude the transmission of any further messages from this process during this cycle. (As a result, at most one **REQUEST** per cycle can be sent from any process.)

Sending an **ACK** causes the req_lock to be set, thereby precluding any subsequent transmission of **REQUEST**s in that cycle. **ACK**s are sent only if a **REQUEST** has not been transmitted (line R3) and only in response to a **REQUEST** (hence, only one per destination). []

Feature 3: No process will receive a message with a cycle number outside the range [cycle_num,cycle_num+1]

Assume the contrary.

Assume that process P_i receives a message from process P_j containing a cycle number cnum<cycle_num. Since cycle numbers increase monotonically, P_i participated in a cycle where cnum was the prevailing cycle_num, but incremented it. Incrementing is done during RESET, immediately after all expected replies have arrived. In particular, a message (**REQUEST** or **ACK**) must have arrived from P_j carrying cnum. As a result of Feature 2, only a single message could have been sent from P_j to P_i during that cycle. We thus have a contradiction.

Assume that P_i has received a message from P_j with cnum>cycle_num+1. This message must be a **REQUEST** since an **ACK** could come only in response to P_i's **REQUEST** which has not yet used a cycle number greater than its cycle_num. P_i, having sent a **REQUEST** with cnum, must have decided that the cycle in which cycle_num+1 was the prevailing cycle number had terminated, implying that an **ACK** from P_j had arrived. This is a contradiction because P_j could not have transmitted a **REQUEST** with a cycle number greater than cycle_num (line S3). []

Corollary: Only the current and next cycles need be identified and thus cycle numbers can be counted modulo 2 (1 bit)!

Feature 4: All cycles end.

A cycle ends when a process receives all the messages belonging to that cycle. By Feature 2, it is sufficient to count the replies as there will be at most one reply from any other user. Counting is done by decrementing a counter every time a regular message arrives (lines R5, R6) and watching when the counter reaches 0. We distinguish two cases, depending on whether or not this process has sent a **REQUEST**.

A process is never blocked in sending a response. Upon receiving a **REQUEST** it immediately sends an **ACK** unless it has previously sent a **REQUEST**. This response eventually arrives at the requestor. Thus, if a **REQUEST** is transmitted all N-1 responses eventually arrive. The counter that has been set to N-1 (line S6) will be decremented N-1 times and reach 0 at the end of the cycle.

If this process did not send a **REQUEST** in the current cycle, it replies with **ACK** to all k arriving **REQUEST**s, decrementing the counter k times. The arrival of the COMPLETE message increments the counter by k, resulting in a zero counter (a cycle always starts with a zero counter). []

Feature 5: The algorithm is deadlock free

Deadlock is a situation in which a process waits indefinitely to enter the critical section. As a result of Feature 4 all cycles end

and, because writing into the next is permitted if immediate **REQUEST** transmittal is not (line S1), a process is guaranteed to be able to transmit its **REQUEST** and enter the critical section thereafter. []

Feature 6: Group mutual exclusion is achieved

Group mutual exclusion is achieved if all group members know one another and if new groups are formed one at a time. Since each cycle ends (Feature 4) all messages must have arrived. The arriving **REQUEST** messages (marked in the current array) correspond to the group members.

New cycles start only after RESET, which is executed when all messages have arrived (lines S7, R8), i.e., when the cycle ends. Consequently, a new cycle starts only after the previous one has ended. []

C. THE LOOP-BASED-TREE MUTUAL-EXCLUSION (LBT-ME) ALGORITHM

In this appendix we describe a mutual-exclusion algorithm operating on a netwrok with a loop-based-tree topology. A loop-based tree is a topology with a single directed arc emanating from each node; the topology consists of exactly one loop such that if an arc belonging to the loop is removed a tree results. The LBT-ME algorithm differs from the standard mutual exclusion algorithm in that the nodes do not all know one another; each node knows only its immediate neighbor, and the network topology is simplified.

1. Description

The algorithm starts by having each node choose a single other node (called its successor) and send it a message containing the sending node's unique ID. Messages with high IDs are forwarded by each node to its successor, until one node receives a message it has already seen. This node then becomes the leader.

A few remarks concerning this algorithm

- To become a leader, a node needs to receive a message it has already seen and not necessarily originated because, in the loop based tree topology only a node on the loop can possibly see the same message twice.
- This is a mutual exclusion and not an election algorithm as the leader is the only one that knows it was elected. To inform all other nodes the leader' identity, each node, starting with the leader, can send to all its predecessors a termination message carrying the leader's identity; because of the tree structure this message will propagate efficiently to all nodes.
- The message traversing the loop first (thereby determining the leader) is not necessarily the one originating at the node with the highest ID.

2. Specification

It is assumed that all messages carry unique IDs in them. The following rules are obeyed by each node:

1. It originates a single message to a destination of its choice (called its successor).
2. It records the highest ID it observes in any of the messages it handles, including its own. (We refer to this as the 'recorded ID'.)
3. It forwards to its successor all messages it receives with an ID higher than its recorded ID.

4. It discards all received messages with an ID lower then its recorded ID.

5. A node receiving a message with an ID equal to its recorded ID becomes the leader (which then ceases to forward messages).

3. Proof of Features

Feature 1: Rules R1 through R5 are comprehensive

There is only a single kind of message traversing the graph in a single direction (rules 1 and 3). Rules 3-5 govern the handling of all possible messages. []

Feature 2: The leader resides on the loop

This is trivially true, since other nodes will not receive their messages back (as is required by rule 5). []

Feature 3: At most one leader is selected.

Assume the contrary, i.e., two leaders A and B are selected. Assume further, that A's ID is higher then B's. Consider the events that led to this situation. By feature 2, both A and B reside on the loop and must therefore have seen each other's message. If B's message passed A after the latter transmitted its message, B's message would not have been forwarded (Rule 4) and thus B could not become the leader. If B's message passed A before it transmitted its message, B's message would arrive at B before A's. Thus B becomes the leader and would not forward A's message (Rule 5), which means that A could not become the leader. In either case we have a contradiction. []

As a result of Feature 3, and because each node originates a message, a leader will be elected. Feature 1 provides the necessary argument for assuring the correctness of the algorithm.

4. Configurations Without Unique IDs

To adapt the algorithm to an environment without unique IDs, we make the following changes in the original algorithm:

- The original algorithm is considered a single round, and is repeated r times (r is a predetermined constant).
- Messages carry both a round number and an ID randomly generated by each node (a different one for every round).
- The pair (round-number, random-ID) is used instead of IDs for comparisons under all rules.

o A node that has been elected all r times is considered a leader.

This algorithm still has a probability of terminating incorrectly, i.e., with more than one leader elected. Obviously, if two or more nodes on the loop generate the same ID, more than one leader will be elected. There is a nonzero probability that this would reccur all r rounds.

It is also possible that a leader not on the loop will be chosen. Consider two nodes on a tree branch, one a successor of the other, choosing the same random ID. The second of the two, upon receiving the message of the first, thinks it is its own message and completes the round. For this to happen in r rounds requires that at least r nodes on the same branch of a tree choose the same number in the first round, that all the leaders (at least r-1 of them) again choose the same number, and that finally in the r-1st round, two nodes select the same number. A branch must contain at least k+r-1 nodes to generate k leaders in r rounds.

It should be noted that since unique IDs are not available, destinations may not be unique; therefore the topology generated is not necessarily a loop based tree. This fact decreases the efficiency of the algorithm (more messages are sent) while increasing the probability of failure.

REFERENCES

1. J. F. Shoch, "Internetwork Naming, Addressing, and Routing," in Proc. 17th IEEE Comp. Soc. Int. Conf. (CompCon), (September 1978).

2. J. H. Saltzer, "On the Naming and Binding of Network Destinations," pp. 311-317 in Proc. of the International Symposium on Local Computer Networks, ed. P. C. Ravasio and N. Naffah, IFIP TC-6, Florence, Italy (April 1982).

3. H. Zimmerman, "OSI Reference Model-The ISO Model for Open System Interconnection," IEEE Trans. on Communications, Vol. COM-28, (4)(April 1980).

4. A. Segall and M. Sidi, "A Failsafe Distributed Protocol for Minimum Delay Routing," IEEE Trans. on Communications, Vol. 29, (5) pp. 689-695 (May 1981).

5. J. M. McQuillan, I Richer, and E. C. Rosen, "The New Routing Algorithm for the ARPANET," IEEE Trans. on Communications, Vol. 28, (5)(May 1980).

6. R. E. Kahn, S. A. Gronemeyer, J. Burchfiel, and R. C. Kunzelman, "Advances in Packet Radio Technology," pp. 1468-1496 in Proceedings of the IEEE, (November 1978).

7. D. E. Rubin, "Toward a Military Interoperable Network," SRI Project 2058, SRI International, Menlo Park, California (April 1982).

8. C. Sunshine and J. Postel, "Addressing Mobile Hosts in the ARPA Internet Environment," IEN 138, USC Information Science Institute (March 1980).

9. N. Shacham and K. S. Klemba, On the Organization of Large Packet Radio Networks (Draft), SRI International, Menlo Park, California (June 1982). [Private Communication]

10. J.B. Postel (ed.), "Internet Protocol," RFC 791, Defense Advanced Research Projects Agency, Information Processing Techniques Office (September 1981).

11. E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," Comm. of the ACM, Vol. 8, (9) p. 569 (September 1965).

12. L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," Comm. of the ACM, Vol. 17, (8) pp. 453-455 (August 1974).

13. G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Comm. of the ACM, Vol. 24, (1) pp. 9-17 (January 1981).

14. R. G. Gallager, P. A. Humblet, and P. M. Spira, "A Distributed Algorithm for Minimum Spanning Tree," LIDS-P-906-A, M.I.T., Cambridge, Mass. (October 1979).

15. H. Garcia-Molina, "Elections in a Distributed Computing System," IEEE Trans. on Computers, Vol. C-31, (1) pp. 48-59 (January 1982).

16. A. Itai and M. Rodeh, "Symmetry Breaking in Distributed Networks," pp. 150-158 in Proc. of the 22nd Annual Symposium on the Foundations of Computer Science, IEEE, Nashville, Tennessee (October 1981).

TAB AND INDEX CORRECTION SHEET

TAB NUMBER	AD NUMBER PROCESSED

AD NUMBER	FIELD	CORRECTIVE ACTION

SIGNATURE	DATE

DTIC FORM AUG 80 222

PREVIOUS EDITION IS OBSOLETE